Técnicas Avanzadas de Programación

Práctica Pre Parcial

1. Explica brevemente **la importancia de los algoritmos** en la programación. Menciona al menos tres razones.
2. Define **eficiencia de un algoritmo**.
3. Explica la diferencia entre **complejidad temporal** y **complejidad espacial**.
4. ¿Qué es el **análisis de mejor caso, peor caso y caso promedio** en un algoritmo? Da un ejemplo de cada uno usando el algoritmo de búsqueda lineal.
5. Explica brevemente el **análisis empírico** y **análisis teórico** de algoritmos.
6. A continuación tienes un fragmento de código python

```python
def busqueda(lista, valor):
    for i in range(len(lista)):
        if lista[i] == valor:
            return i
    return -1
```

   - Describe en qué escenario este algoritmo tiene el mejor caso y el peor caso.
   - Calcula la **complejidad temporal** del algoritmo.
7. Escribe un algoritmo en **Python** que implemente **búsqueda lineal**, donde el usuario debe ingresar una lista de números y el valor a buscar.
8. Implementa en **Python** una versión optimizada de la búsqueda lineal que detenga la búsqueda si la lista está ordenada y se detecta un valor mayor al buscado.
9. Elige entre los siguientes dos algoritmos de búsqueda: **búsqueda lineal** y **búsqueda binaria**. Implementa el más eficiente para buscar un número en una lista ordenada y explica por qué lo elegiste.
10. **Define qué es el análisis de algoritmos** y menciona por qué es importante en el desarrollo de software eficiente.
11. Explica las diferencias entre **análisis teórico** y **análisis empírico** de un algoritmo.
12. Define la **notación Big O** y su propósito en el análisis de algoritmos.
13. Proporciona un ejemplo de un algoritmo que tenga una complejidad de **O(n)** y explica por qué tiene esa complejidad.
14. ¿Qué significa que un algoritmo tenga una complejidad de **$O(n^2)$**? Da un ejemplo de un algoritmo que presente esta complejidad y explica su funcionamiento.
15. ¿Qué es el **análisis de mejor caso, peor caso y caso promedio**? Explica estos conceptos utilizando el algoritmo de **búsqueda binaria** como ejemplo.
16. Define la **notación Big O** y su propósito en el análisis de algoritmos.
17. Proporciona un ejemplo de un algoritmo que tenga una complejidad de **O(n)** y explica por qué tiene esa complejidad.
18. ¿Qué significa que un algoritmo tenga una complejidad de **$O(n^2)$**? Da un ejemplo de un algoritmo que presente esta complejidad y explica su funcionamiento.
19. Calcula la complejidad temporal de este algoritmo y justifica tu respuesta:

```python
def suma_numeros(lista):
    total = 0
    for numero in lista:
        total += numero
    return total
```

20. Implementa en **Python** el algoritmo de **búsqueda binaria** para buscar un valor en una lista de números ordenados
    a. Explica por qué la **búsqueda binaria** es más eficiente que la **búsqueda lineal** en listas ordenadas.

21. Implementa en **Python** un algoritmo que calcule la suma de los primeros **n** números enteros. Calcula la complejidad temporal de tu algoritmo.
22. **Define qué es la recursión** y menciona las dos partes fundamentales de un algoritmo recursivo.
23. Diferencia entre **recursión directa** y **recursión indirecta**, proporcionando un ejemplo de cada una.
24. Menciona dos **ventajas** y dos **desventajas** de la recursión frente a las soluciones iterativas.
25. Escribe el algoritmo recursivo del **factorial** de un número. Explica cómo funciona cada llamada recursiva.
26. Explica cómo funciona la **recursión en el algoritmo de Fibonacci**. Cual es su **complejidad temporal** y explica por qué no es eficiente para valores grandes de **n**.
27. A continuación se muestra un código que implementa la **recursión** para calcular el n-ésimo término de la serie de Fibonacci. Identifica el caso base y el caso recursivo y explica la complejidad del algoritmo:

```
14  def fibonacci(n):
15      if n == 0 or n == 1:
16          return n
17      return fibonacci(n-1) + fibonacci(n-2)
```

28. A partir del siguiente código recursivo de **búsqueda binaria**, explica cómo funciona la recursión en este algoritmo y calcula su complejidad:

```
19  def busqueda_binaria(lista, inicio, fin, valor):
20      if inicio > fin:
21          return -1
22      medio = (inicio + fin) // 2
23      if lista[medio] == valor:
24          return medio
25      elif lista[medio] > valor:
26          return busqueda_binaria(lista, inicio, medio - 1, valor)
27      else:
28          return busqueda_binaria(lista, medio + 1, fin, valor)
```

29. Implementa en **Python** una función recursiva que **invierta una cadena de texto**. La función debe recibir una cadena y devolverla invertida.
30. Implementa en **Python** el cálculo de la **potencia de un número** usando recursión.
31. **Explica cómo funciona la recursión en la Serie de Fibonacci** y menciona por qué la implementación recursiva básica no es eficiente.
32. Define el algoritmo de **Merge Sort** y describe su funcionamiento. Explica por qué su complejidad temporal es **O(n log n)**.
33. Diferencia entre **Merge Sort** y **Quick Sort** en cuanto a eficiencia.

34. Identifique el algoritmo, describe su funcionamiento. Explica por qué su complejidad temporal

```python
def sort(lista):
    if len(lista) <= 1:
        return lista

    mitad = len(lista) // 2
    izquierda = sort(lista[:mitad])
    derecha = sort(lista[mitad:])
    return unir(izquierda, derecha)

def unir(izquierda, derecha):
    resultado = []
    i = j = 0
    while i < len(izquierda) and j < len(derecha):
        if izquierda[i] < derecha[j]:
            resultado.append(izquierda[i])
            i += 1
        else:
            resultado.append(derecha[j])
            j += 1
    resultado += izquierda[i:]
    resultado += derecha[j:]

    return resultado
```

35. Identifique el algoritmo, describe su funcionamiento. Explica por qué su complejidad temporal

```python
def sort(lista):
    if len(lista) <= 1:
        return lista
    menores, pivote, mayores = particionar(lista)
    return sort(menores) + [pivote] + sort(mayores)

def particionar(lista):
    pivote = lista[len(lista) // 2]
    menores = []
    mayores = []
    for elemento in lista:
        if elemento < pivote:
            menores.append(elemento)
        elif elemento > pivote:
            mayores.append(elemento)
    return menores, pivote, mayores
```

36. ¿Qué es la **búsqueda binaria** recursiva? Explica cómo funciona el algoritmo y proporciona un ejemplo en Python para encontrar un número en una lista ordenada.
37. Explica la diferencia entre **recursión directa** y **recursión indirecta**, y proporciona un ejemplo en Python de cada una.
38. Implementa en **Python** una versión recursiva del algoritmo **Merge Sort**.
39. Implementa en **Python** el algoritmo **Quick Sort** para ordenar una lista de números
40. **Define qué es la recursión de cola** y explica por qué es más eficiente que la recursión tradicional en términos de uso de memoria.
41. Diferencia entre **memoización** y **tabulación** en la programación dinámica. Da un ejemplo de cada técnica.

42. Explica la importancia de la **optimización de la recursión** en problemas complejos.
43. Implementa en **Python** el cálculo de Fibonacci utilizando **memoización**. Explica por qué este enfoque es más eficiente que la implementación recursiva básica.
44. Implementa el algoritmo de Fibonacci utilizando **tabulación**. indique su **complejidad temporal** y explica por qué es más eficiente que la versión recursiva estándar sin optimización.
45. Implementa en python una función factorial con **recursión de cola**. Explica cómo funciona el algoritmo.
46. En la siguiente implementación de Fibonacci, identifica cómo se aplica la **memoización** y por qué mejora el rendimiento. Indica también la **complejidad temporal** del algoritmo:

```python
def fibonacci_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n == 0 or n == 1:
        memo[n] = n
    else:
        memo[n] = fibonacci_memo(n-1, memo) + fibonacci_memo(n-2, memo)
    return memo[n]
```

47. **Define el algoritmo de búsqueda lineal** y menciona en qué tipo de situaciones es más eficiente su uso.
48. Explica brevemente el **algoritmo de ordenamiento burbuja (Bubble Sort)** y por qué su complejidad temporal es **$O(n^2)$**.
49. compara y explica los algoritmos de **inserción** y **selección** en términos de eficiencia y funcionamiento. Proporciona ejemplos de cuándo sería mejor usar cada uno.
50. Calcula la **complejidad temporal** del siguiente algoritmo de **búsqueda binaria** y explica cómo funciona: (BigO)

```python
def busqueda_binaria(lista, valor):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == valor:
            return medio
        elif lista[medio] < valor:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1
```

51. Proporciona un análisis de **mejor caso**, **peor caso** y **caso promedio** para el algoritmo de **búsqueda secuencial**.

52. Analiza el siguiente código de **ordenamiento,** identifica el algoritmo, explica su funcionamiento y su análisis de complejidad temporal.

```
95  def sort(lista):
96      for i in range(len(lista)):
97          min_idx = i
98          for j in range(i + 1, len(lista)):
99              if lista[j] < lista[min_idx]:
100                 min_idx = j
101         lista[i], lista[min_idx] = lista[min_idx], lista[i]
```

53. Analiza el siguiente código de **ordenamiento** e identifica el algoritmo, explica su funcionamiento. ¿Cuál es la **complejidad temporal** de este algoritmo en el mejor y peor de los casos?

```
103 def sort(lista):
104     for i in range(1, len(lista)):
105         valor_actual = lista[i]
106         j = i - 1
107         while j >= 0 and lista[j] > valor_actual:
108             lista[j + 1] = lista[j]
109             j -= 1
110         lista[j + 1] = valor actual
```

54. Implementa el algoritmo de **ordenamiento burbuja (Bubble Sort) standard** en **Python** para ordenar una lista de números enteros.
    - Explica por qué este algoritmo es ineficiente para listas grandes y menciona su complejidad temporal.
    - Optimizarlo para que tenga peor caso y caso promedio
55. Implementa el algoritmo de **búsqueda binaria** en una lista ordenada de números enteros.